

The Role of Vertex Consistency in Sampling-based Algorithms for Optimal Motion Planning

Oktay Arslan*

Panagiotis Tsiotras†

Abstract

Motion planning problems have been studied by both the robotics and the controls research communities for a long time, and many algorithms have been developed for their solution. Among them, incremental sampling-based motion planning algorithms, such as the Rapidly-exploring Random Trees (RRT), and the Probabilistic Road Maps (PRM) have become very popular recently, owing to their implementation simplicity and their advantages in handling high-dimensional problems. Although these algorithms work very well in practice, the quality of the computed solution is often not good, i.e., the solution can be far from the optimal one. A recent variation of RRT, namely the RRT* algorithm, bypasses this drawback of the traditional RRT algorithm, by ensuring asymptotic optimality as the number of samples tends to infinity. Nonetheless, the convergence rate to the optimal solution may still be slow. This paper presents a new incremental sampling-based motion planning algorithm based on Rapidly-exploring Random Graphs (RRG), denoted RRT[#] (RRT “sharp”) which also guarantees asymptotic optimality but, in addition, it also ensures that the constructed spanning tree of the geometric graph is consistent after each iteration. In consistent trees, the vertices which have the potential to be part of the optimal solution have the minimum cost-come-value. This implies that the best possible solution is readily computed if there are some vertices in the current graph that are already in the goal region. Numerical results compare with the RRT* algorithm.

Keywords: optimal motion planning, RRT, RRG, RRT*, RRT[#], vertex consistency, consistent tree.

1 Introduction

Motion planning problems are crucial for the realization of truly autonomous vehicles and robots. Many approaches have been proposed in the literature (see for example, the excellent books by LaValle [15] and Choset et al [1]). A bottleneck in most motion planning problems, especially those involving systems with high state dimensionality, is the computational overhead associated with discretizing (i.e., gridding) the state space. Hence, deterministic searches [5] are impractical for high dimensional state spaces. Probabilistic roadmap methods [2, 10, 22, 11], [1, Ch. 7], as well as methods that use rapidly exploring random trees (RRTs) [16, 17, 6, 3, 21], are among the most popular. They can address the vehicle’s kinematic and dynamic constraints during motion planning in high dimensional state spaces. In these methods, random samples of the obstacle-free space are connected to each other by feasible trajectories, and the resulting graph is searched for a sequence of connected samples from the initial state to the goal state. Sampling-based algorithms

*Oktay Arslan is a graduate student with the D. Guggenheim School of Aerospace Engineering, Georgia Institute of Technology, Atlanta, GA 30332-0150, USA, Email: oktay@gatech.edu

†Professor Panagiotis Tsiotras is with the faculty of D. Guggenheim School of Aerospace Engineering, Georgia Institute of Technology, Atlanta, GA 30332-0150, USA, Email: tsiotras@gatech.edu

require efficient low-level collision detection and trajectory planning algorithms to find collision-free trajectories between different samples [17].

Incremental sampling-based algorithms were first proposed by Kavraki during the late 1990s. The so-called Probabilistic Road Map (PRM) was successfully implemented to solve multi-query motion planning problems and gained a lot of attention, both in industry and academia [12]. In PRM a graph of the environment is constructed by taking random samples from the configuration space of the robot and testing them to determine whether they belong to the free space. The PRM algorithm uses a local planner that attempts to find a feasible path between the sampled points. Once a reasonable graph is constructed, the initial and the goal states are added to the graph, and the optimal path is computed using a graph search algorithm.

Another important class of incremental sampled-based motion planning algorithm is the Rapidly-exploring Random Tree (RRT) and its numerous variants [17]. RRTs have achieved great success in solving single-query motion planning problems in many real-time applications. However, the quality of RRT-based algorithms is often poor (i.e., highly suboptimal). As a result, a lot of effort has been devoted to the development of heuristic techniques in order to refine the quality of the solution obtained from RRTs. However, it has been recently shown that the best path returned by RRTs when the algorithm converges is almost always (i.e., with probability one) far from optimal [9]. This has renewed the interest to develop incremental sampled-based algorithms for motion-planning problems with optimality guarantees. In [7] the authors proposed the Rapidly-exploring Random Graphs (RRG) algorithm, which has asymptotic optimality properties, that is, it ensures that the optimal path will be found as the number of samples tends to infinity. Based on RRG, the same authors later proposed a new algorithm, namely RRT* that extracts a tree from the graph constructed by RRG [8, 9].

In this paper we present a new incremental sampling-based motion planning algorithm based on RRG, denoted RRT[#] (RRT “sharp”), which also guarantees asymptotic optimality but, in addition, it also ensures that, at each step, the constructed spanning tree of the graph is consistent. Vertex consistency (see Section 2) implies that the accumulated cost-to-come of each vertex equals to the optimal cost-to-come. This allows us classify the vertices according to their potential of being part of the optimal path, and thus to quickly identify the region where the optimal solution is more likely to be found. This information can be subsequently used to improve the speed of convergence of the standard RRT* algorithm, as well as in order to more efficiently explore the obstacle-free space. Three variants of the baseline RRT[#] algorithm are proposed that take advantage of this vertex classification to speed up convergence.

The organization of the paper is as follows: The problem formulation is given in the next section. In Section 3, an overview of the RRT[#] algorithm is introduced. The fundamental concepts and primitive functions used in the RRT[#] algorithm are explained. In Section 4, each step of the proposed approach is explained in detail, along with the pseudo-code of the algorithm and the main procedures used in the main algorithm. In Sections 5, simulation results are used to compare the solutions of the proposed approach with the well-known RRT* algorithm. In Section 6, several variants of the baseline algorithm are presented by using simple vertex rejection techniques and improvements are demonstrated by doing extensive simulations in the subsequent section. We conclude the paper with some possible extensions for future work.

2 Problem Formulation

2.1 Notation and Definitions

Let \mathcal{X} denote the state space, which is assumed to be an open subset of \mathbb{R}^d , where $d \in \mathbb{N}$ with $d \geq 2$. Let the *obstacle region* and the *goal region* be denoted by \mathcal{X}_{obs} and $\mathcal{X}_{\text{goal}}$, respectively. The obstacle-free space is defined by $\mathcal{X}_{\text{free}} = \mathcal{X} \setminus \mathcal{X}_{\text{obs}}$. Let the *initial state* be denoted by $x_{\text{init}} \in \mathcal{X}_{\text{free}}$. The neighborhood of a state $x \in \mathcal{X}$ is defined as the open ball of radius $r \in \mathbb{R}_+$ centered at x , that is, $B_r(x) = \{x' \in \mathcal{X} : \|x - x'\| < r\}$. Let $\mathcal{G} = (V, E)$ denote a graph, where V and $E \subseteq V \times V$ are finite sets of vertices and edges, respectively. In the sequel, we will use graphs to represent the connections between a (finite) set of points selected randomly from $\mathcal{X}_{\text{free}}$. With a slight abuse of notation, we will use x to denote both the point in the space \mathcal{X} and the corresponding vertex in the graph.

Geometric r -disc graph: Let $V \subset \mathbb{R}^d$ be a finite set, and $r \geq 0$. A geometric r -disc graph $\mathcal{G}(V; r) = (V, E)$ in d dimensions is an undirected graph with vertex set V and edge set $E = \{(u, v) : u, v \in V \text{ and } \|u - v\| < r\}$.

Successor vertices: Given a vertex $v \in V$, the set-valued function $\text{succ} : (\mathcal{G}, v) \mapsto V' \subseteq V$ returns the vertices in V that can be reached from vertex v ,

$$\text{succ}(\mathcal{G}, v) := \{u \in V : (v, u) \in E\}$$

Predecessor vertices: Given a vertex $v \in V$ in a directed graph $\mathcal{G} = (V, E)$, the function $\text{pred} : (\mathcal{G}, v) \mapsto V' \subseteq V$ returns the vertices in V that are the tails of the edges going into v ,

$$\text{pred}(\mathcal{G}, v) := \{u \in V : (u, v) \in E\}$$

Parent vertex: Given a vertex $v \in V$, the function $\text{parent} : v \mapsto u$ returns the unique vertex $u \in V$ such that $(u, v) \in E$ and $u \in \text{pred}(\mathcal{G}, v)$.

Spanning tree: Given the graph $\mathcal{G} = (V, E)$, a spanning tree of \mathcal{G} can be defined such that $\mathcal{T} = (V_s, E_s)$, where $V_s = V$ and $E_s = \{(u, v) : u, v \in V, (u, v) \in E \text{ and } \text{parent}(v) = u\}$.

Edge cost value: Given an edge $e = (u, v) \in E$, the function $\mathbf{c} : e \mapsto r$ returns a non-negative real number. Then $\mathbf{c}(u, v)$ where $v \in \text{succ}(\mathcal{G}, u)$ is the cost incurred by moving from u to v . *Cost-to-come value:* Given a vertex $v \in V$, the function $\mathbf{g} : v \mapsto r$ returns a non-negative real number r , which is the cost of the path to v from a given initial state $x_{\text{init}} \in \mathcal{X}_{\text{free}}$. Let $\mathbf{g}^*(v)$ be the optimal cost-to-come value of the vertex v . The optimal cost-to-come satisfies the following relationship:

$$\mathbf{g}^*(v) = \begin{cases} 0, & \text{if } v = x_{\text{init}}, \\ \min_{u \in \text{pred}(\mathcal{G}, v)} (\mathbf{g}^*(u) + \mathbf{c}(u, v)), & \text{otherwise.} \end{cases}$$

Each vertex v is associated with two estimates of the optimal cost-to-come value $\mathbf{g}^*(v)$, namely, $\mathbf{g}(v)$ (g-value) and $\text{lmc}(v)$ (locally minimum cost-to-come estimate, or lmc-value). The $\text{lmc}(v)$ is the best estimate of the cost-to-come of the vertex v , computed based on the g-value of the vertices in the predecessor set $\text{pred}(v)$. The lmc-value (also called rhs-value in [13]) is a one-step ahead lookahead value based on the g-value and is thus potentially better informed than the g-value of the vertex. The lmc-value satisfies the following relationship

$$\text{lmc}(v) = \begin{cases} 0, & \text{if } v = x_{\text{init}}, \\ \min_{u \in \text{pred}(\mathcal{G}, v)} (\mathbf{g}(u) + \mathbf{c}(u, v)), & \text{otherwise.} \end{cases}$$

Heuristic value: Given a vertex $v \in V$, and a goal region $\mathcal{X}_{\text{goal}}$, the function $\mathbf{h} : (v, \mathcal{X}_{\text{goal}}) \mapsto r \in \mathbb{R}$ returns an estimate of the optimal cost from v to $\mathcal{X}_{\text{goal}}$; it is 0 if $v \in \mathcal{X}_{\text{goal}}$. It is an admissible

heuristic if it never overestimates the actual cost of reaching $\mathcal{X}_{\text{goal}}$. In this paper, we always assume an admissible heuristic. It is well known that inadmissible heuristics can be used to speed-up the algorithm, but they lead to suboptimal paths [19].

Relevant region: Let $x_{\text{goal}}^* \in \mathcal{X}_{\text{goal}}$ be the point in the goal region that has the lowest optimal cost-to-come value in $\mathcal{X}_{\text{goal}}$, i.e., $x_{\text{goal}}^* = \operatorname{argmin}_{x \in \mathcal{X}_{\text{goal}}} g^*(x)$. The *relevant region* of $\mathcal{X}_{\text{free}}$ is the set of points x for which the optimal cost-to-come value of x , plus the estimate of the optimal cost moving from x to $\mathcal{X}_{\text{goal}}$ is less than the optimal cost-to-come value of x_{goal}^* , that is,

$$\mathcal{X}_{\text{rel}} = \{x \in \mathcal{X}_{\text{free}} : g^*(x) + h(x) < g^*(x_{\text{goal}}^*)\}$$

Points that lie in the \mathcal{X}_{rel} have the potential to be part of the optimal path starting at x_{init} and reaching $\mathcal{X}_{\text{goal}}$.

Key value: Given a vertex $v \in V$, the function $\text{Key} : v \mapsto k$ returns a real vector $k \in \mathbb{R}^2$, whose components are $k_1(v) = \min(g(v), \text{lmc}(v)) + h(v)$ and $k_2(v) = \min(g(v), \text{lmc}(v))$, respectively. Components of the keys correspond to the f-values and g-values in the A* algorithm, respectively [18].

Promising vertices: Let $v_{\text{goal}}^* \in V$ be the vertex that has the lowest key value, i.e., $v_{\text{goal}}^* = \operatorname{argmin}_{v \in V \cap \mathcal{X}_{\text{goal}}} \text{Key}(v)$. The *promising vertices* $V_{\text{prom}} \subseteq V$ is the set of vertices that have better key value than v_{goal}^* , that is,

$$V_{\text{prom}} = \{v \in V : \text{Key}(v) \prec \text{Key}(v_{\text{goal}}^*)\}$$

Priority of vertices: The priority of vertices in the queue is the same as the priority of their associated keys, and the precedence relation between keys is determined according to lexicographical ordering. Given two keys $k, k' \in \mathbb{R}^2$, the Boolean function $\prec : (k, k') \mapsto \{\text{False}, \text{True}\}$ returns **True** if and only if either $k_1 < k'_1$ or $(k_1 = k'_1 \text{ and } k_2 \leq k'_2)$, and **False** otherwise.

Consistency: A vertex $v \in V$ is called *locally consistent* if and only if its g-value equals its lmc-value [13]. Otherwise, it is an *inconsistent* vertex. The notion of *consistency* is very important because it allows one to update cost-to-come values of all vertices by propagating the effects of the changes in the topology of the graph. This way, an incremental search can reuse information from the previous searches, thus speeding up the whole algorithm. The lmc-value always keeps the best up-to-date estimate of the cost-to-come value based on the current topology of the graph, whereas the g-value keeps an estimate of the cost-to-come value computed from a previous topology of the graph. Equality of the g- and lmc-values of a vertex implies that the changes in the topology of the graph will not effect the cost-to-come value of that vertex, that is, the topology of the graph is consistent with its previous configuration in the locality of the vertex.

A tree $\mathcal{T} = (V_s, E_s)$ is called a *consistent tree* if and only if all of its promising vertices are consistent.

The g-value of all vertices equals to their respective optimal cost-to-come value if and only if all vertices are locally consistent [13]. The g-values have the following form when all vertices are locally consistent

$$g(v) = \begin{cases} 0, & \text{if } v = x_{\text{init}}, \\ \min_{u \in \text{pred}(\mathcal{G}, v)} (g(u) + c(u, v)), & \text{otherwise.} \end{cases}$$

Then, the shortest path from $x_{\text{init}} \in \mathcal{X}_{\text{free}}$ to any vertex $v \in V$ can be found by starting at v and traversing iteratively from the current vertex $u \in V$ to any of its predecessor $u' \in \text{pred}(\mathcal{G}, u)$ that minimizes $g(u') + c(u', u)$ (ties can be broken arbitrarily), until x_{init} is reached.

2.2 Problem Definition

The proposed RRT[#] algorithm solves the following motion planning problem: Given a bounded and connected open set $\mathcal{X} \subset \mathbb{R}^d$, and the sets $\mathcal{X}_{\text{free}}$ and $\mathcal{X}_{\text{obs}} = \mathcal{X} \setminus \mathcal{X}_{\text{free}}$, and given an initial point $x_{\text{init}} \in \mathcal{X}_{\text{free}}$ and a goal region $\mathcal{X}_{\text{goal}} \subset \mathcal{X}_{\text{free}}$, find the minimum-cost path connecting x_{init} to the goal region $\mathcal{X}_{\text{goal}}$. If no such path exists, then report that no solution is possible.

3 The RRT[#] Algorithm - Overview

A brief description of each function used in the RRT[#] algorithm is given below.

Sampling: **Sample** : $\mathbb{N} \rightarrow \mathcal{X}_{\text{free}}$ is a function that returns independent, identically distributed (i.i.d) samples from $\mathcal{X}_{\text{free}}$.

Nearest neighbor: **Nearest** is a function that returns a point from a given finite set V , which is the closest to a given point x in terms of a given distance function.

Near vertices: **Near** is a function that returns n number of points from a given finite set V , which is the closest to a given point x in terms of a given distance function.

Steering: **Steer** is a function that returns the closest point in a ball centered around a given state x to another given point x_{new} .

Collision checking: Given two points, the Boolean function **ObstacleFree** checks whether the minimum distance path connecting these two points belongs to $\mathcal{X}_{\text{free}}$. It returns **True** if the line segment is a subset of the $\mathcal{X}_{\text{free}}$.

Tree extension: **Extend** is a function that extends the nearest vertex of the tree \mathcal{T} towards the randomly sampled point x_{rand} .

Reducing inconsistency: Given a graph $\mathcal{G} = (V, E)$, a corresponding spanning tree $\mathcal{T} = (V_s, E_s)$, where $V_s = V$ and $E_s \subset V \times V$ and a goal region $\mathcal{X}_{\text{goal}} \subset \mathcal{X}_{\text{free}}$, the function **ReduceInconsistency** : $(\mathcal{G}, \mathcal{T}, \mathcal{X}_{\text{goal}}) \mapsto (\mathcal{G}, \mathcal{T}')$ operates on the inconsistent vertices of the tree \mathcal{T} iteratively, and continues until the tree becomes consistent, that is, all vertices of the tree that are promising (see Section 4) are consistent. The **ReduceInconsistency** function is used to propagate the effects of the topological changes in the graph \mathcal{G} as new vertices are added with each iteration.

A priority queue is used to sort all of the inconsistent vertices of the tree \mathcal{T} based on their respective key values. The following functions are defined to manage the priority queue.

Update queue: Given a vertex $v \in V$, the function **UpdateQueue** changes the content of the queue based on the g- and lmc-values of the vertex v . If the vertex v is inconsistent, then it is either inserted into the queue or its priority in the queue is updated based on its up-to-date key value if it is already inside the queue. Otherwise, the vertex is removed from the queue if it is a consistent vertex.

Find minimum: The function **findmin()** returns the vertex with the highest priority of all vertices in the queue, i.e., the vertex of minimum key value.

Remove a vertex: Given a vertex $v \in V$, the function **remove()** deletes the vertex v from content of the queue.

Update priority: Given a vertex $v \in V$, and a key value k , the function **update()** changes the priority of the vertex v in priority queue q , i.e., it reassigns the key value of the vertex v with the new given key value k .

Inserting a vertex: Given a vertex $v \in V$, and a key k , the function **insert()** adds the vertex v with the key value k into queue.

4 The RRT[#] Algorithm - Details

The body of the RRT[#] algorithm is given in Algorithm 1 and it is similar to the other RRT-variants (RRT, RRG, RRT*, etc) with the notable exception that it keeps track of vertex consistency using the key values of all current vertices in the graph. One of the important difference between the RRT* and RRT[#] algorithms is that all vertices in the tree computed by the RRT* algorithm have a uniform type based on their finite cost-to-come value, whereas in the RRT[#] algorithm the vertices have different types based on their pair of estimates of the cost-to-come value. In the RRT[#] algorithm, each vertex v can be classified in one of the following four categories based on the values of its $(g(v), lmc(v))$ pair.

- Consistent with finite key value: $g(v) < \infty, lmc(v) < \infty$ and $g(v) = lmc(v)$
- Consistent with infinite key value: $g(v) = \infty, lmc(v) = \infty$
- Inconsistent with finite key value: $g(v) < \infty, lmc(v) < \infty$ and $g(v) \neq lmc(v)$
- Inconsistent with infinite g-value and finite lmc-value: $g(v) = \infty, lmc(v) < \infty$

Vertices in the second category are always non-promising, whereas vertices in the rest of categories can be either promising or non-promising. The promising vertices can be used to approximate the region $\mathcal{X}_{rel} \subseteq \mathcal{X}_{free}$ of the free space that may contain the optimal path.

Algorithm 1: Body of the RRT[#] Algorithm

```

1 RRT#( $x_{init}, \mathcal{X}_{goal}, \mathcal{X}$ )
2    $V \leftarrow \{x_{init}\}; E \leftarrow \emptyset;$ 
3    $\mathcal{G} \leftarrow (V, E);$ 
4   for  $i = 1$  to  $N$  do
5      $x_{rand} \leftarrow \text{Sample}(i);$ 
6      $\mathcal{G} \leftarrow \text{Extend}(\mathcal{G}, x_{rand});$ 
7      $\text{ReduceInconsistency}(\mathcal{G}, \mathcal{X}_{goal});$ 
8    $(V, E) \leftarrow \mathcal{G}; E' \leftarrow \emptyset;$ 
9   foreach  $x \in V$  do
10     $E' \leftarrow E' \cup \{(\text{parent}(x), x)\}$ 
11  return  $\mathcal{T} = (V, E')$ 

```

The algorithm starts by adding the initial point x_{init} into the vertex set of the underlying graph. Then, it incrementally grows the graph in \mathcal{X}_{free} by sampling a random point x_{rand} from \mathcal{X}_{free} and extending some parts of the graph towards x_{rand} . Later, the **ReduceInconsistency** procedure, which is provided in Algorithm 3, propagates the new information due to the extension across the whole graph in order to improve the estimate of the cost-to-come value of the promising vertices in the graph. All computations due to the sampling and extension steps, followed by information propagation (Lines 4-7 of Algorithm 1), form a single *iteration* of the algorithm. The process is repeated for a given fixed number of iterations, and the consistent spanning tree of the final graph is returned at the end.

The key difference between the RRT[#] algorithm and other RRT-variants is that a unique consistent spanning tree of the graph is maintained at the end of the each iteration of the algorithm. Since this tree is consistent, it contains information of the lowest-cost path, which can be achieved on the current graph, for each promising vertex of the graph. In addition, the g-value of the promising

vertices equals to their respective optimal cost-to-come value that can be achieved through the edges of the tree. Therefore, each new vertex is initialized with the minimum possible estimate of its respective optimal cost-to-come value during extension (since all of its promising neighbor vertices have the lowest g-value), and this estimate keeps improving to the best possible value whenever new information becomes available on any part of the graph. Hence, the g-value of each promising vertex of the graph converges to its optimal cost-to-come value very quickly.

Algorithm 2: Extend Procedure for RRT[#] Algorithm

```

1 Extend( $\mathcal{G}, x$ )
2    $(V, E) \leftarrow \mathcal{G}; E' \leftarrow \emptyset;$ 
3    $x_{\text{nearest}} \leftarrow \text{Nearest}(\mathcal{G}, x);$ 
4    $x_{\text{new}} \leftarrow \text{Steer}(x_{\text{nearest}}, x);$ 
5   if  $\text{ObstacleFree}(x_{\text{nearest}}, x_{\text{new}})$  then
6      $g(x_{\text{new}}) \leftarrow \infty;$ 
7      $\text{lmc}(x_{\text{new}}) = g(x_{\text{nearest}}) + c(x_{\text{nearest}}, x_{\text{new}});$ 
8      $\text{parent}(x_{\text{new}}) = x_{\text{nearest}};$ 
9      $\mathcal{X}_{\text{near}} \leftarrow \text{Near}(\mathcal{G}, x_{\text{new}}, |V|);$ 
10    foreach  $x_{\text{near}} \in \mathcal{X}_{\text{near}}$  do
11      if  $\text{ObstacleFree}(x_{\text{near}}, x_{\text{new}})$  then
12        if  $\text{lmc}(x_{\text{new}}) > g(x_{\text{near}}) + c(x_{\text{near}}, x_{\text{new}})$  then
13           $\text{lmc}(x_{\text{new}}) = g(x_{\text{near}}) + c(x_{\text{near}}, x_{\text{new}});$ 
14           $\text{parent}(x_{\text{new}}) = x_{\text{near}};$ 
15         $E' \leftarrow E' \cup \{(x_{\text{near}}, x_{\text{new}}), (x_{\text{new}}, x_{\text{near}})\};$ 
16     $V \leftarrow V \cup \{x_{\text{new}}\};$ 
17     $E \leftarrow E \cup E';$ 
18     $\text{UpdateQueue}(x_{\text{new}});$ 
19  return  $\mathcal{G}' \leftarrow (V, E)$ 

```

The **Extend** procedure for the RRT[#] algorithm is given in Algorithm 2. During each iteration, the **Extend** procedure tries to extend the graph towards the randomly sampled point $x_{\text{rand}} \in \mathcal{X}_{\text{free}}$. First, the closest vertex in the graph x_{nearest} is found in Line 3, then x_{nearest} is steered towards the randomly sampled point x_{rand} in the next line. If the line segment connecting the steered point x_{new} and x_{nearest} is feasible, then the new point x_{new} is prepared for inclusion to the vertex set of the graph. First, its cost-to-come estimate, i.e., the g-value and lmc-values, and the parent vertex are initialized by using information of the nearest vertex x_{nearest} . Then, a local search is performed in some neighborhood of x_{new} , i.e., the set of vertices returned by the **Near** procedure, in order to find the local minimum cost-to-come estimate value in Lines 10-15 and the corresponding parent vertex. The new vertex x_{new} and all extensions resulting in feasible trajectories are added to the vertex and edge set of the graph in Lines 16-17, respectively. In the end, the new vertex is decided to be inserted in the priority queue or not based on its consistency in the **UpdateQueue** procedure.

Inclusion of each new vertex may result in an inconsistent vertex in the graph if a finite lmc-value is achieved. Therefore, consistency of the spanning tree needs to be checked, and appropriate operations must be performed in order to make it consistent, if necessary. The **ReduceInconsistency** procedure, which is provided in Algorithm 3, is called to make the spanning tree consistent by operating on the inconsistent and *promising* vertices of the graph, iteratively. It simply pops the most promising inconsistent vertex from the priority queue, if there are any, and this inconsistent vertex is made consistent by assigning its lmc-value to its g-value. Then, its new g-value information is propagated among its neighbors in order to improve their lmc-values in Lines 7-11. However,

Algorithm 3: ReduceInconsistency Procedure

```

1 ReduceInconsistency( $\mathcal{G}, \mathcal{X}_{\text{goal}}$ )
2   while  $q.\text{findmin}() \prec \text{Key}(x_{\text{goal}}^*)$  do
3      $x = q.\text{findmin}()$ ;
4      $g(x) = \text{lmc}(x)$ ;
5      $q.\text{delete}(x)$ ;
6     foreach  $s \in \text{succ}(\mathcal{G}, x)$  do
7       if  $\text{lmc}(s) > g(x) + c(x, s)$  then
8          $\text{parent}(s) = x$ ;
9          $\text{lmc}(s) = g(x) + c(x, s)$ ;
10         $\text{UpdateQueue}(s)$ ;

```

Algorithm 4: Auxiliary Procedures

```

1 Initialize( $x$ )
2    $g(x) \leftarrow \infty$ ;
3    $\text{lmc}(x) \leftarrow \infty$ ;
4    $\text{parent}(x) \leftarrow \emptyset$ ;
5 UpdateQueue( $x$ )
6   if  $g(x) \neq \text{lmc}(x)$  and  $x \in q$  then
7      $q.\text{update}(x, \text{Key}(x))$ ;
8   else if  $g(x) \neq \text{lmc}(x)$  and  $x \notin q$  then
9      $q.\text{insert}(x, \text{Key}(x))$ ;
10  else if  $g(x) = \text{lmc}(x)$  and  $x \in q$  then
11     $q.\text{delete}(x)$ ;
12 Key( $x$ )
13    $g_{\min} = \min(g(x), \text{lmc}(x))$ ;
14    $f = g_{\min} + h(x)$ ;
15   return  $key = (f, g_{\min})$ ;

```

this information propagation may also cause some vertices to be inconsistent; therefore, all resulting inconsistent vertices are inserted in the priority queue as well. This process continues until a consistent spanning tree is computed, that is, there is no inconsistent promising vertex left in the priority queue.

5 Numerical Simulations 1

The RRT[#] algorithm was developed in C++ and run on a computer with a 2.40 GHz processor and 12GB RAM running the Ubuntu 11.10 Linux operating system. A Fibonacci heap was implemented as priority queue to store inconsistent vertices during the search [4]. Extensive simulations were run to compare the performance of the RRT[#] algorithm with the RRT* algorithm, whose C implementation is available to download from the RRT* authors' website (<http://sertac.scripts.mit.edu/rrtstar/>).

Both RRT[#] and RRT* algorithms were run on three different problem types with the same sample sequence in order to demonstrate the difference in their behavior while growing the tree. All problems tested require finding an optimal path in a square environment minimizing the Euclidean

path length. The heuristic value of a vertex is the Euclidean distance from the vertex to the goal. In the first problem type, there are no obstacles in the environment, whereas there are some box-like obstacles in the second and third problem types. In the third problem type, the environment is more cluttered than the one in the second problem type, containing many widely distributed small obstacles.

For the first problem type, the trees computed by both algorithms at different stages are shown in Figure 1. The initial state is plotted as a yellow square and the goal region is shown in blue with magenta border (upper right). The minimal-length path is shown in red. As shown in Figure 1, the best path computed by the $RRT^\#$ algorithm converges to the optimal path. As mentioned earlier, one of the important differences between the RRT^* and $RRT^\#$ algorithms is that the latter classifies the vertices in one of the following four categories based on the values of its $(g(v), lmc(v))$ pair: Consistent with finite key value (shown in green), consistent with infinite key value (shown in black), inconsistent with finite key value (shown in blue), and inconsistent with infinite g-value and finite lmc-value (shown in red).

Since only the points in the relevant region \mathcal{X}_{rel} have the potential to be part of the optimal path, the $RRT^\#$ algorithm tries to approximate \mathcal{X}_{rel} with the set of promising vertices V_{prom} and tends to stop rewiring the parts of the tree which lie outside of the \mathcal{X}_{rel} as iterations go to infinity. As seen in Figure 1, for this particular scenario, \mathcal{X}_{rel} is an elliptic region, which is much smaller than the whole \mathcal{X}_{free} . Therefore, uniform random sampling on \mathcal{X}_{free} results in too many vertices of different types (green, black, red, and blue vertices) outside of the relevant region during the search. The estimate of \mathcal{X}_{rel} can be used to implement more intelligent sampling strategies, if needed, although this possibility was not pursued in this paper, where all sampling was uniform.

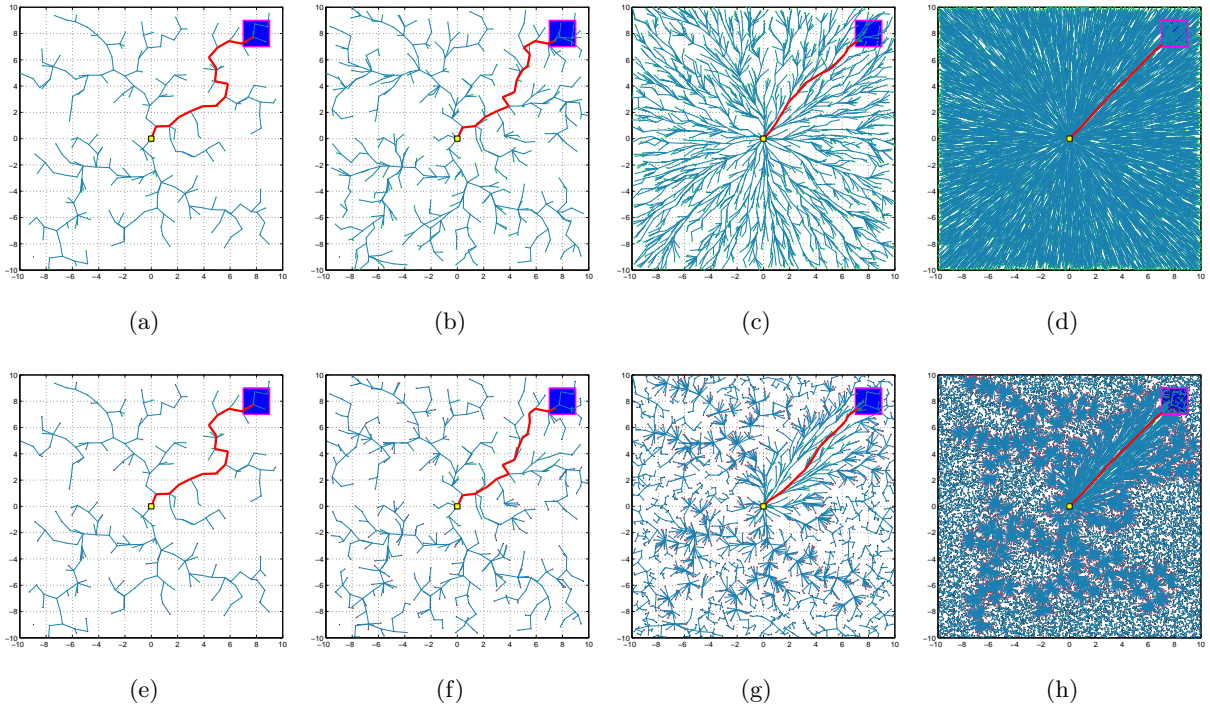


Figure 1: The evolution of the tree computed by RRT^* and $RRT^\#$ algorithms is shown in (a)-(d) and (e)-(h), respectively. The configuration of the trees (a), (e) is at 250 iterations, (b), (f) is at 500 iterations, (c), (g) is at 2500 iterations, and (d), (h) is at 25000 iterations.

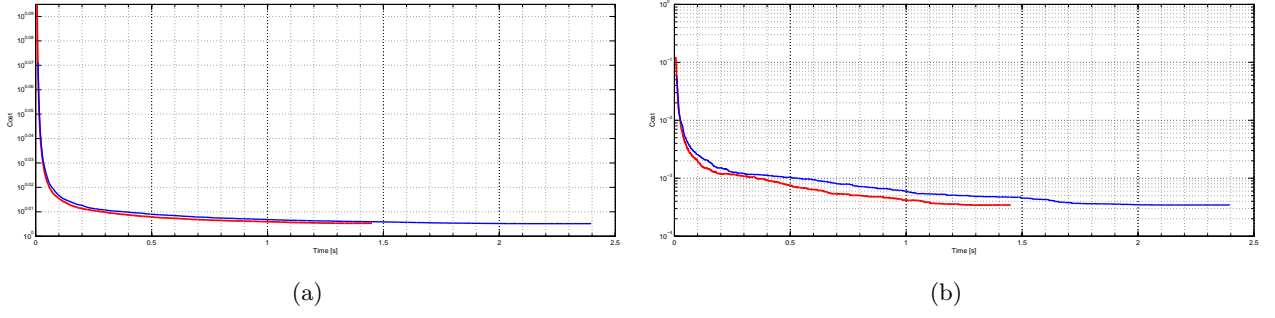


Figure 2: The change in the cost of the best paths computed by RRT* and RRT# algorithms, and the variance in the trials are shown in (a) and (b), respectively.

In the second problem type, the same experiment was carried out and both algorithms were run in an environment with several obstacles. The configuration of the trees for both the RRT* and RRT# algorithms at different stages are shown in Figure 3.

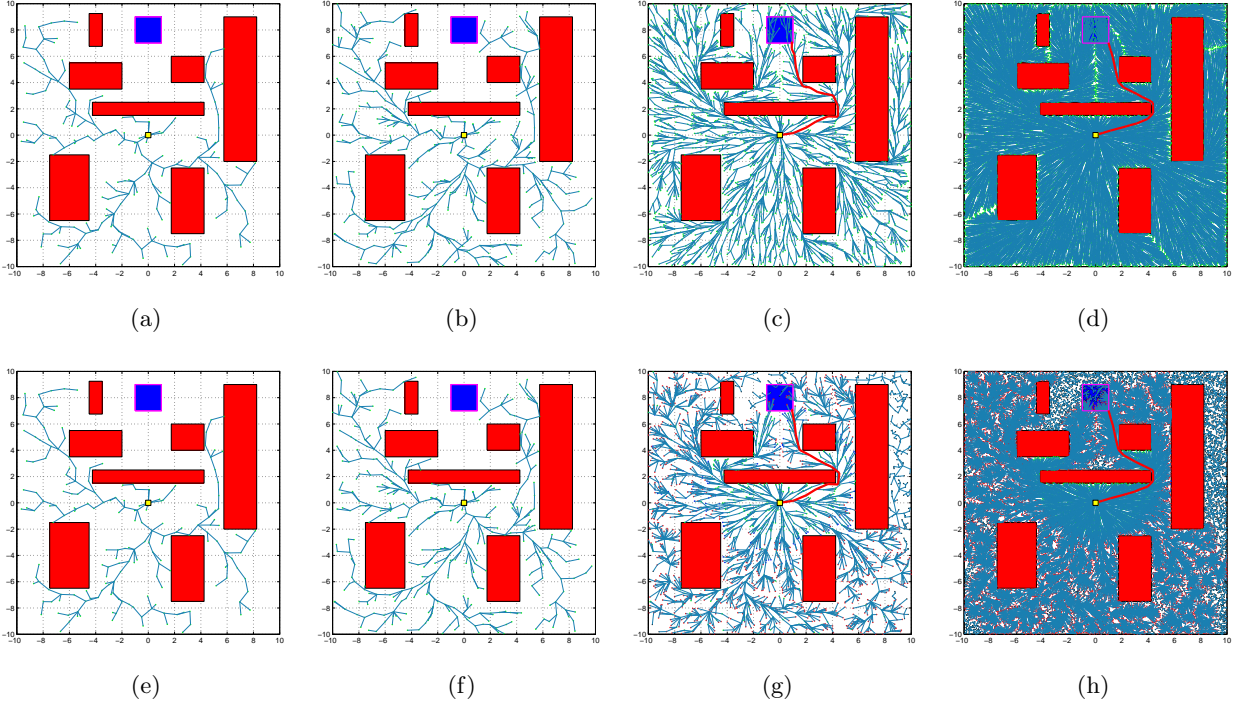


Figure 3: The evolution of the tree computed by RRT* and RRT# algorithms is shown in (a)-(d) and (e)-(h), respectively. The configuration of the trees (a), (e) is at 250 iterations, (b), (f) is at 500 iterations, (c), (g) is at 2500 iterations, and (d), (h) is at 25000 iterations.

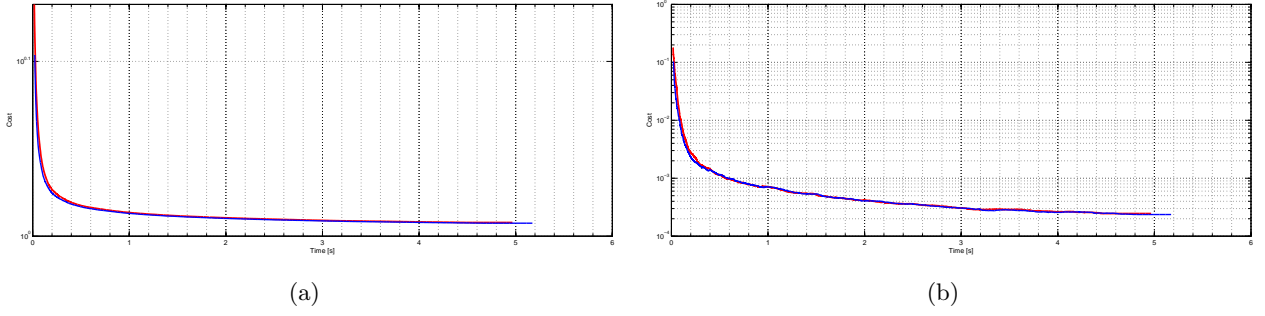


Figure 4: The change in the cost of the best paths computed by RRT* and RRT[#] algorithms and the variance in the trials are shown in (a) and (b), respectively.

In the third problem type, both algorithms were run in a more cluttered environment, where there are many different homotopy classes containing the local minimum solution for the problem. As shown in Figure 5, both algorithms switch between paths which have locally best cost, eventually converging to the optimal solution.

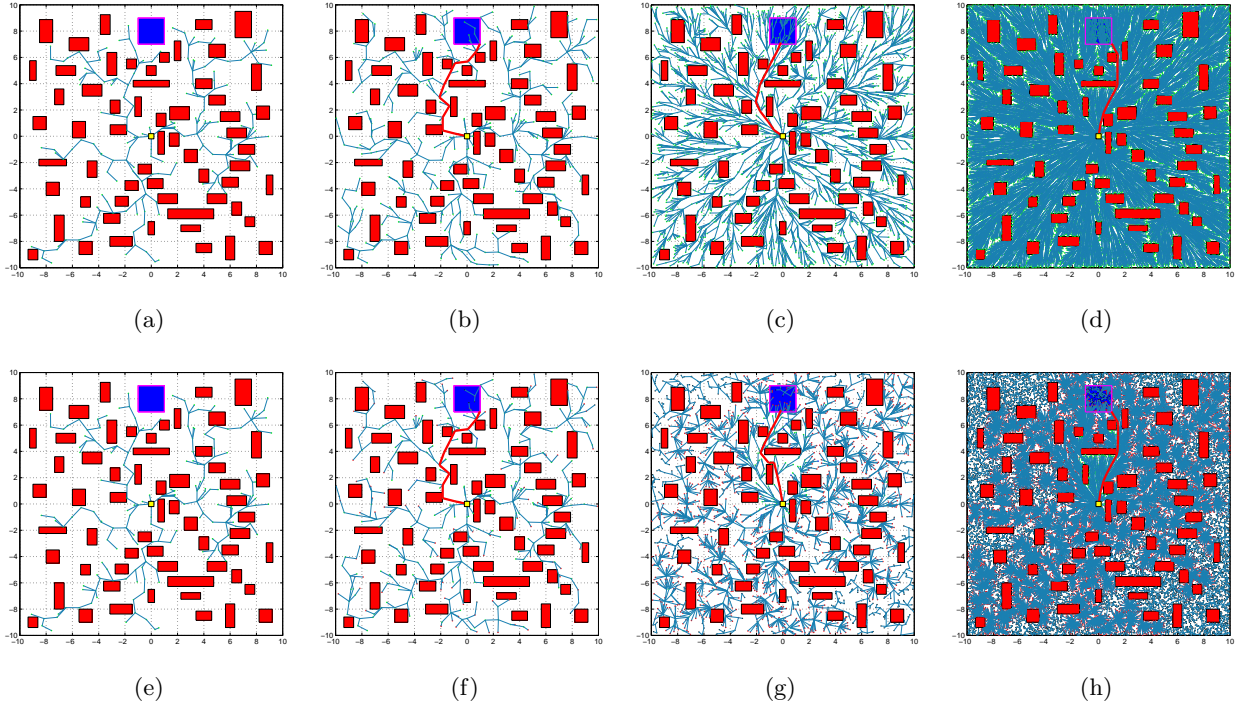


Figure 5: The evolution of the tree computed by RRT* and RRT[#] algorithms is shown in (a)-(d) and (e)-(h), respectively. The configuration of the trees (a), (e) is at 250 iterations, (b), (f) is at 500 iterations, (c), (g) is at 2500 iterations, and (d), (h) is at 25000 iterations.

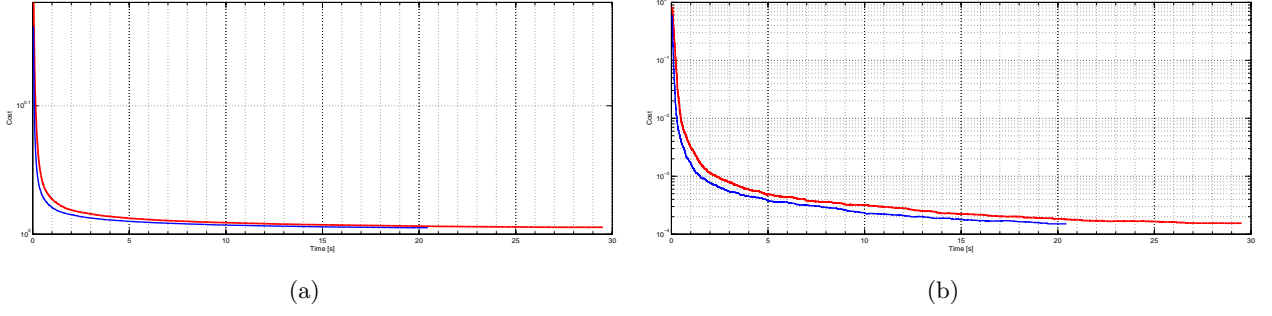


Figure 6: The change in the cost of the best paths computed by RRT^* and $\text{RRT}^\#$ algorithms and the variance in the trials are shown in (a) and (b), respectively.

Finally, in the fourth problem type, both algorithms were run in a obstacle-free environment where there are different cost zones. The cost coefficient of each zone from top to bottom is 1.5, 0.75, 2.5, 0.75, and 1.5, respectively and 1 elsewhere. As seen in Figure 7, both algorithms compute the optimal path which has longer segments in low-cost zones.

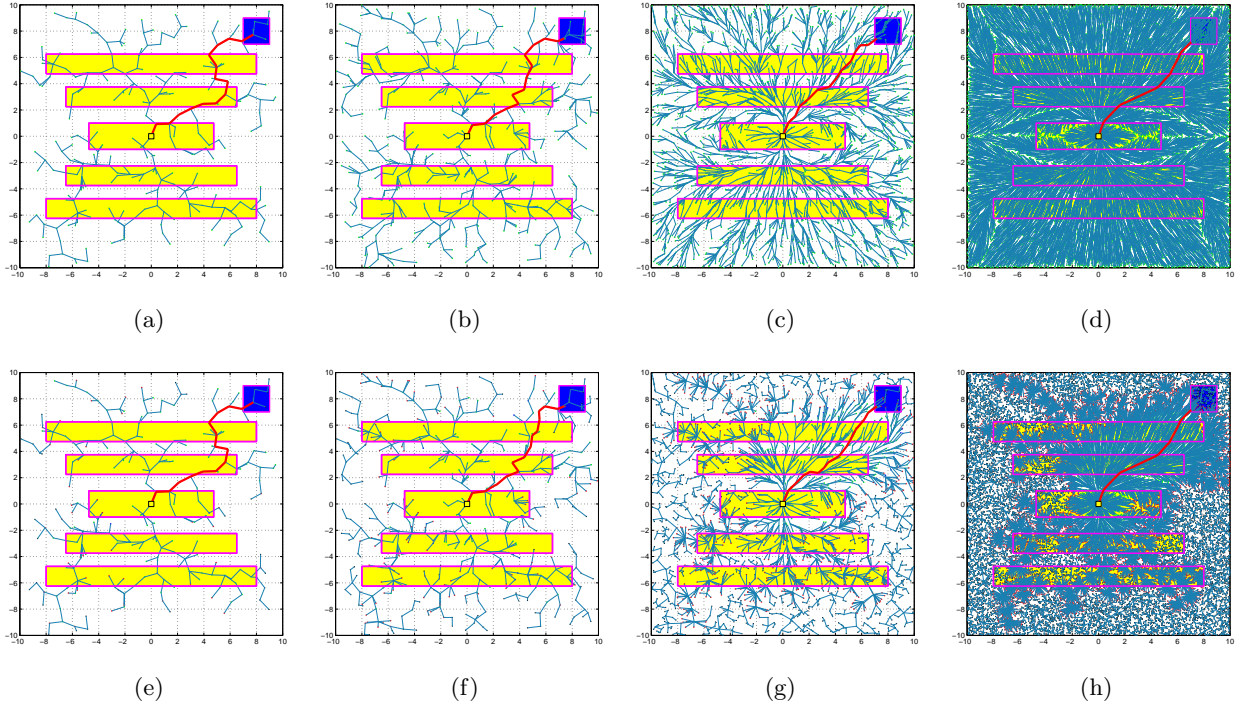


Figure 7: The evolution of the tree computed by RRT^* and $\text{RRT}^\#$ algorithms is shown in (a)-(d) and (e)-(h), respectively. The configuration of the trees (a), (e) is at 250 iterations, (b), (f) is at 500 iterations, (c), (g) is at 2500 iterations, and (d), (h) is at 25000 iterations.

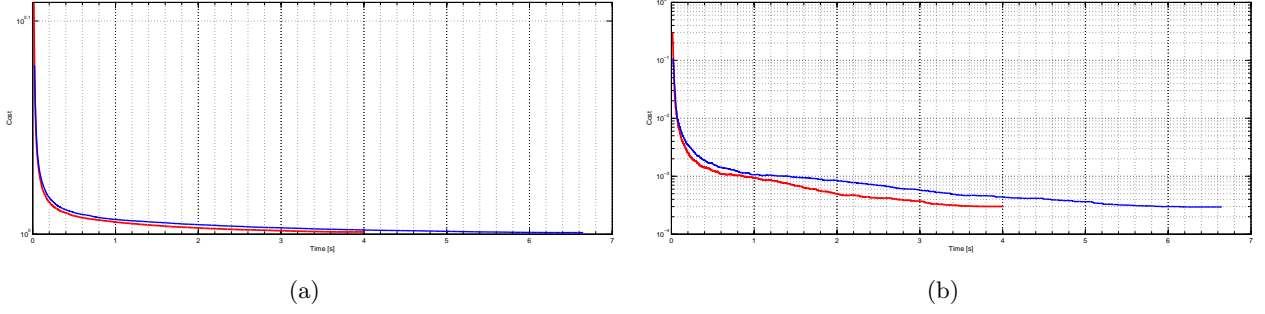


Figure 8: The change in the cost of the best paths computed by RRT* and RRT[#] algorithms, and the variance in the trials are shown in (a) and (b), respectively.

6 Variants of the RRT[#] Algorithm

Too many non-promising vertices are included in the tree computed by the RRT[#] algorithm as observed in the previous simulations. This is owing to the fact that the RRT[#] algorithm includes all new vertices in the graph regardless of their type. A simple vertex selection criterion can be used in the **Extend** procedure in order to prevent the algorithm from growing the tree towards the region outside \mathcal{X}_{rel} . However, being over-selective on vertex inclusion may degrade the performance of the algorithm – and thus lead to a suboptimal solution – since the cost-to-come value of all vertices, which is used to decide if a new vertex is promising or not, is an estimate of the optimal one. In this section, we propose three variants of the baseline RRT[#] algorithm.

RRT[#]_{V1} : In the first variant, which is given in Algorithm 5, if a new vertex happens to be consistent with infinite key value (black vertex), it is not included in the graph. This situation can happen if all of the neighbor vertices of the new vertex happen to be inconsistent with infinite g-value and finite lmc-value (red vertices). First, the estimates of the cost-to-come-value of the new vertex x_{new} are initialized with infinite cost, and its parent vertex is set to ‘null’ in Line 6. Then, a better value for the lmc-value of the new vertex is searched among its neighbor vertices. During this search, the parent of the new vertex remains unassigned only if there are no any neighboring vertices with finite g-value.

RRT[#]_{V2} : In the second variant, the algorithm becomes more selective on vertices to be added to the graph and the “ $\text{parent}(x_{\text{new}}) \neq \emptyset \wedge \text{Key}(\text{parent}(x_{\text{new}})) \prec \text{Key}(x_{\text{goal}}^*)$ ” condition is checked in Line 14. Simply, a new vertex is included to the graph only if its parent is a promising vertex.

RRT[#]_{V3} : Lastly, the third variant is most selective on vertex for inclusion and $\text{Key}(x_{\text{new}}) \prec \text{Key}(x_{\text{goal}}^*)$ condition is checked, that is, only promising new vertices are included in the graph.

Algorithm 5: Extend Procedure for $\text{RRT}_{V_1}^\#$ Algorithm

```

1 Extend( $\mathcal{G}, x$ )
2    $(V, E) \leftarrow \mathcal{G}; E' \leftarrow \emptyset;$ 
3    $x_{\text{nearest}} \leftarrow \text{Nearest}(\mathcal{G}, x);$ 
4    $x_{\text{new}} \leftarrow \text{Steer}(x_{\text{nearest}}, x);$ 
5   if  $\text{ObstacleFree}(x_{\text{nearest}}, x_{\text{new}})$  then
6      $\text{Initialize}(x_{\text{new}});$ 
7      $\mathcal{X}_{\text{near}} \leftarrow \text{Near}(\mathcal{G}, x_{\text{new}}, |V|);$ 
8     foreach  $x_{\text{near}} \in \mathcal{X}_{\text{near}}$  do
9       if  $\text{ObstacleFree}(x_{\text{near}}, x_{\text{new}})$  then
10        if  $\text{lmc}(x_{\text{new}}) > \mathbf{g}(x_{\text{near}}) + \mathbf{c}(x_{\text{near}}, x_{\text{new}})$  then
11           $\text{lmc}(x_{\text{new}}) = \mathbf{g}(x_{\text{near}}) + \mathbf{c}(x_{\text{near}}, x_{\text{new}});$ 
12           $\text{parent}(x_{\text{new}}) = x_{\text{near}};$ 
13           $E' \leftarrow E' \cup \{(x_{\text{near}}, x_{\text{new}}), (x_{\text{new}}, x_{\text{near}})\};$ 
14        if  $\text{parent}(x_{\text{new}}) \neq \emptyset$  then
15           $V \leftarrow V \cup \{x_{\text{new}}\};$ 
16           $E \leftarrow E \cup E';$ 
17           $\text{UpdateQueue}(x_{\text{new}});$ 
18  return  $\mathcal{G}' \leftarrow (V, E)$ 

```

7 Numerical Simulations 2

The same experiments as before were carried out for the three variants of the $\text{RRT}^\#$ algorithm. As seen in the figures below, all variants successfully prevent the inclusion of vertices which lie in the unfavorable regions of the search space. As seen in Figures 9(e), 12(e), 15(e), and 18(e), the $\text{RRT}_{V_1}^\#$ algorithm does not include any black vertices in the tree (these are the vertices that are consistent with infinite key value, hence non-promising), but still computes a solution to the problem, which is as good as the one computed by the RRT^* and $\text{RRT}^\#$ algorithms. However, there are still many red (i.e., non-promising and inconsistent with infinite g-value and finite lmc-value) vertices included in the tree. This is owing to the fact that they are never made consistent until the last iteration, since they mostly lie outside of \mathcal{X}_{rel} . Therefore, they remain in the priority queue and need to be sorted during each iteration. This makes the **ReduceInconsistency** procedure slower. In the $\text{RRT}_{V_2}^\#$ algorithm, the number of red vertices included into the tree is reduced by simply enforcing to have a promising parent vertex for the new vertex that is considered for extension. Red vertices are mostly included into the branches of the tree that are formed outside of the \mathcal{X}_{rel} during exploration phase. As seen in Figures 9(v), 12(v), 15(v), and 18(v), the $\text{RRT}_{V_2}^\#$ algorithm tends not to include vertices into the branches of the tree which are very far away from the optimal solution. Lastly, the $\text{RRT}_{V_3}^\#$ algorithm includes a new vertex into the tree only if it is a promising one. Therefore, all vertices in the tree, other than the goal vertices, are either green or blue, which are located around the boundary of \mathcal{X}_{rel} .

The convergence rate and variance in the computation of the best path for all algorithms are shown in Figures 11, 14, 17, and 20. Since this is a two-dimensional problem, the optimal path for each problem type can be computed visually and the cost of the paths for each algorithm is normalized with respect to the cost of the optimal solution. The ratio of the cost of the best path over the optimal cost for the RRT^* , $\text{RRT}^\#$, $\text{RRT}_{V_1}^\#$, $\text{RRT}_{V_2}^\#$, and $\text{RRT}_{V_3}^\#$ algorithms is shown in red, blue, green, magenta, and black colors, respectively.

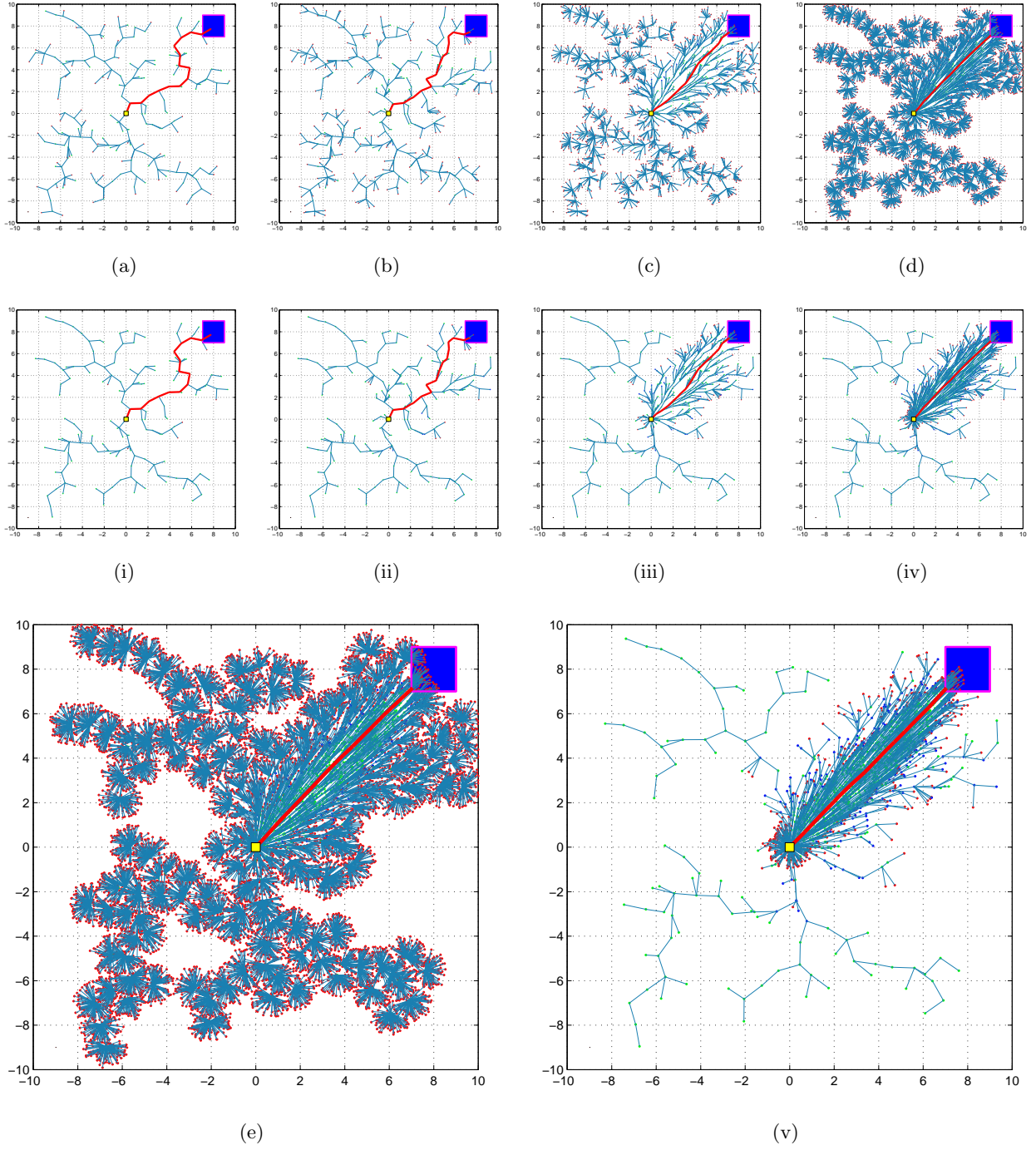


Figure 9: The evolution of the tree computed by $\text{RRT}_{V_1}^\#$ and $\text{RRT}_{V_2}^\#$ algorithms is shown in (a)-(e) and (i)-(v), respectively. The configuration of the trees (a), (i) is at 250 iterations, (b), (ii) is at 500 iterations, (c), (iii) is at 2500 iterations, (d), (iv) is at 10000 iterations, and (e), (v) is at 25000 iterations.

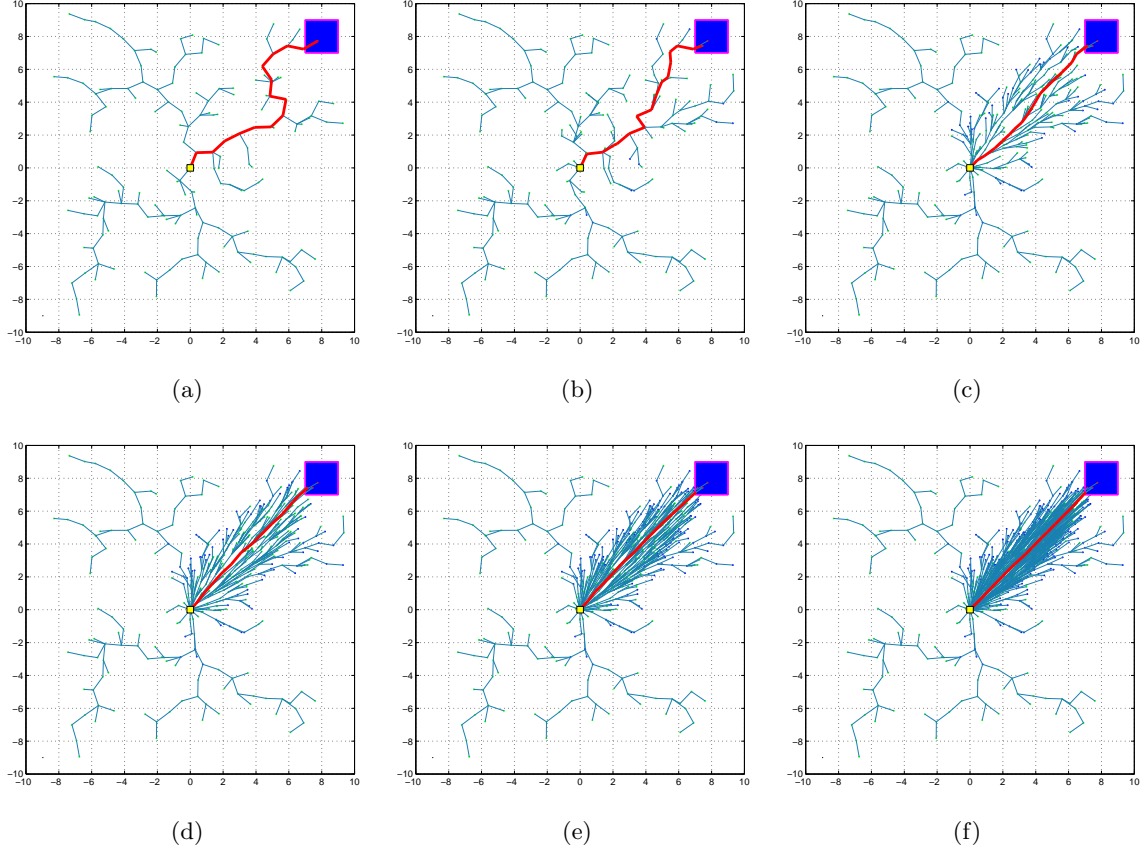


Figure 10: The evolution of the tree computed by $\text{RRT}^{\#}_{V_3}$ algorithm is shown in (a)-(f). The configuration of the trees in (a) is at 250 iterations, in (b) is at 500 iterations, in (c) is at 2500 iterations, in (d) is at 5000 iterations, in (e) is at 10000 iterations, and in (f) is at 25000 iterations.

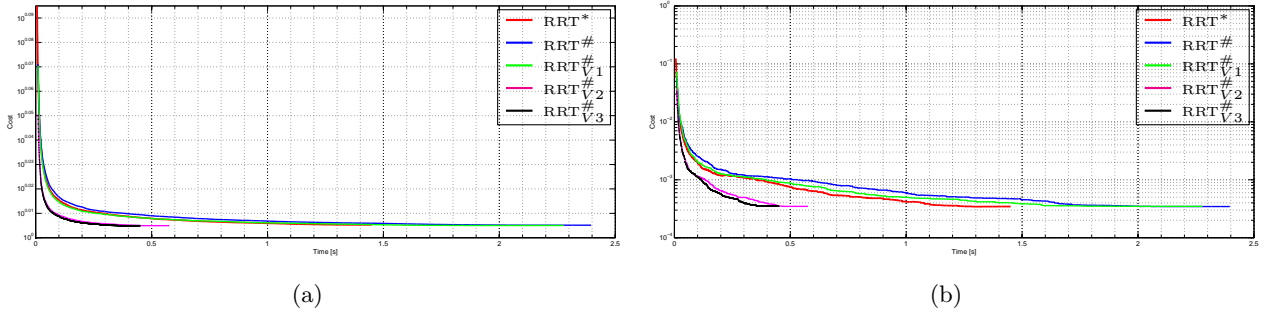


Figure 11: The change in the cost of the best paths computed by RRT^* , $\text{RRT}^{\#}$, and its variant algorithms and the variance in the trials are shown in (a) and (b), respectively.

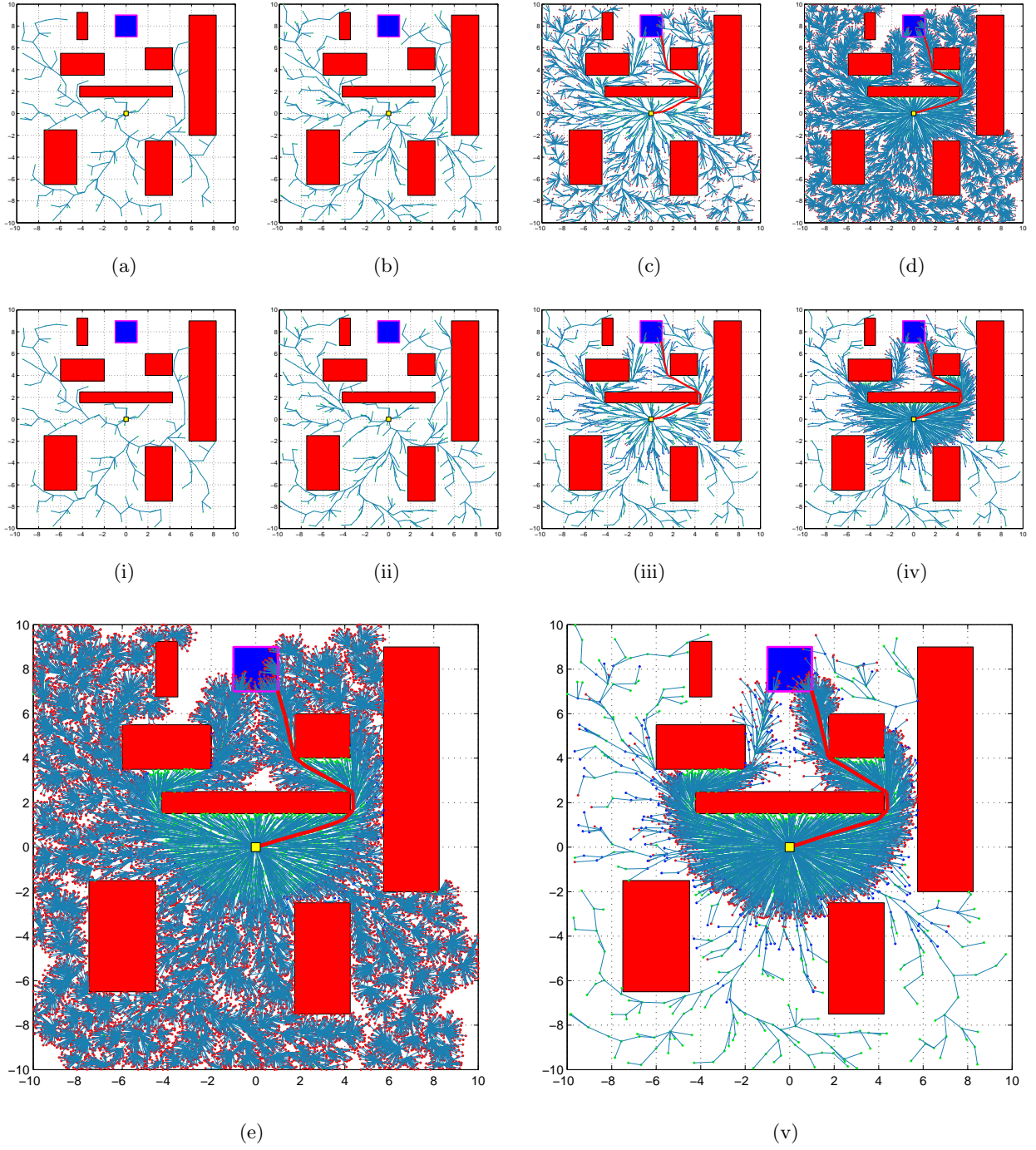


Figure 12: The evolution of the tree computed by $\text{RRT}_{V_1}^\#$ and $\text{RRT}_{V_2}^\#$ algorithms is shown in (a)-(e) and (i)-(v), respectively. The configuration of the trees (a), (i) is at 250 iterations, (b), (ii) is at 500 iterations, (c), (iii) is at 2500 iterations, (d), (iv) is at 10000 iterations, and (e), (v) is at 25000 iterations.

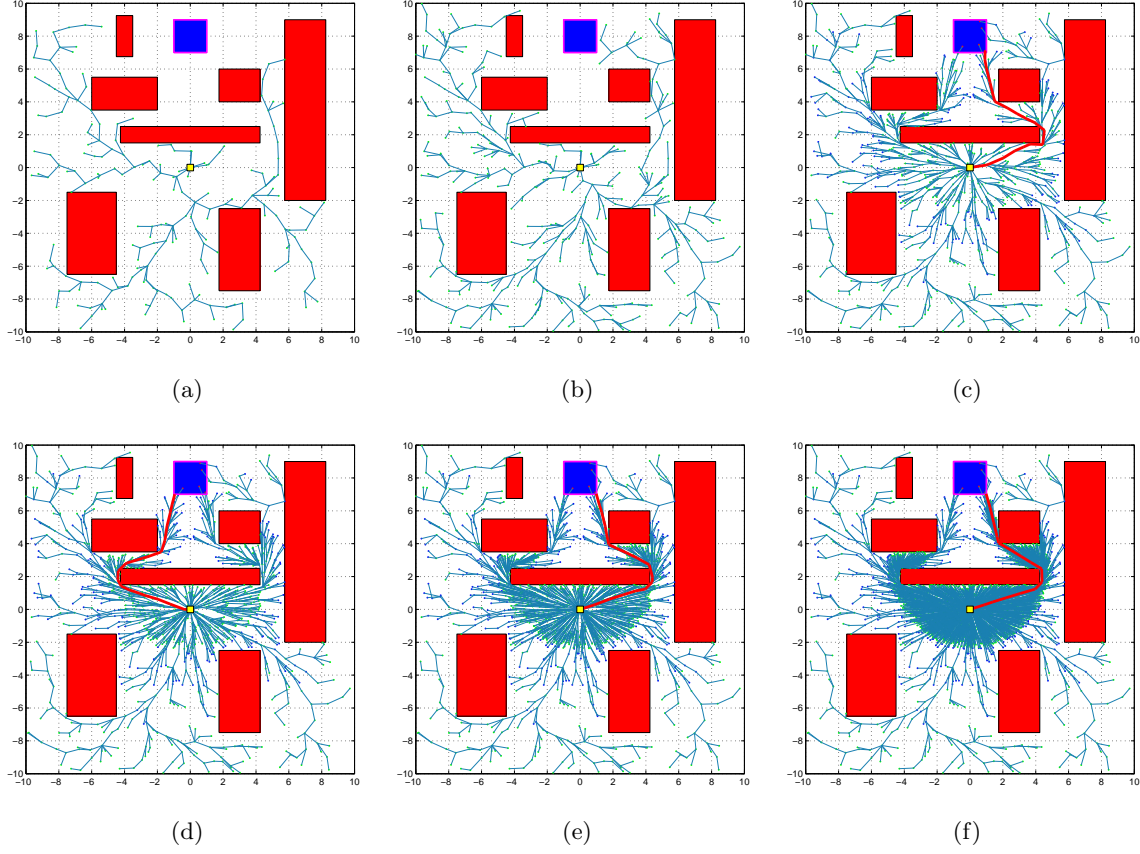


Figure 13: The evolution of the tree computed by $\text{RRT}_{V3}^\#$ algorithm is shown in (a)-(f). The configuration of the trees in (a) is at 250 iterations, in (b) is at 500 iterations, in (c) is at 2500 iterations, in (d) is at 5000 iterations, in (e) is at 10000 iterations, and in (f) is at 25000 iterations.

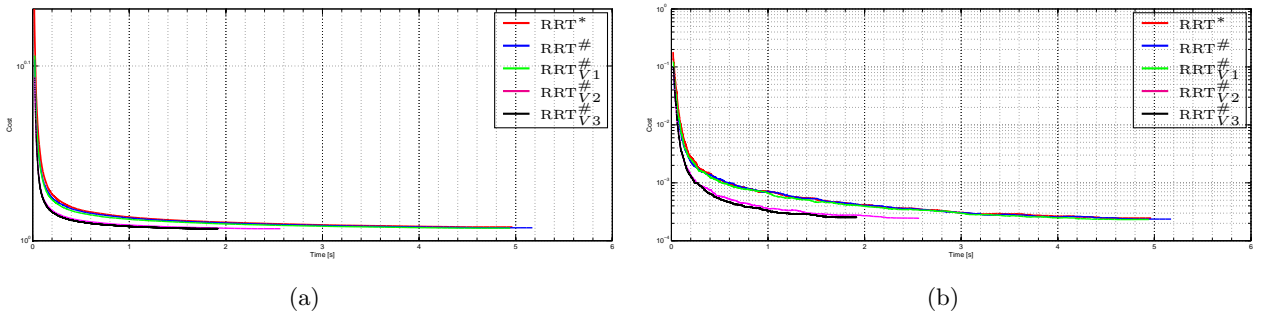


Figure 14: The change in the cost of the best paths computed by RRT^* , $\text{RRT}^\#$, and its variant algorithms and the variance in the trials are shown in (a) and (b), respectively.

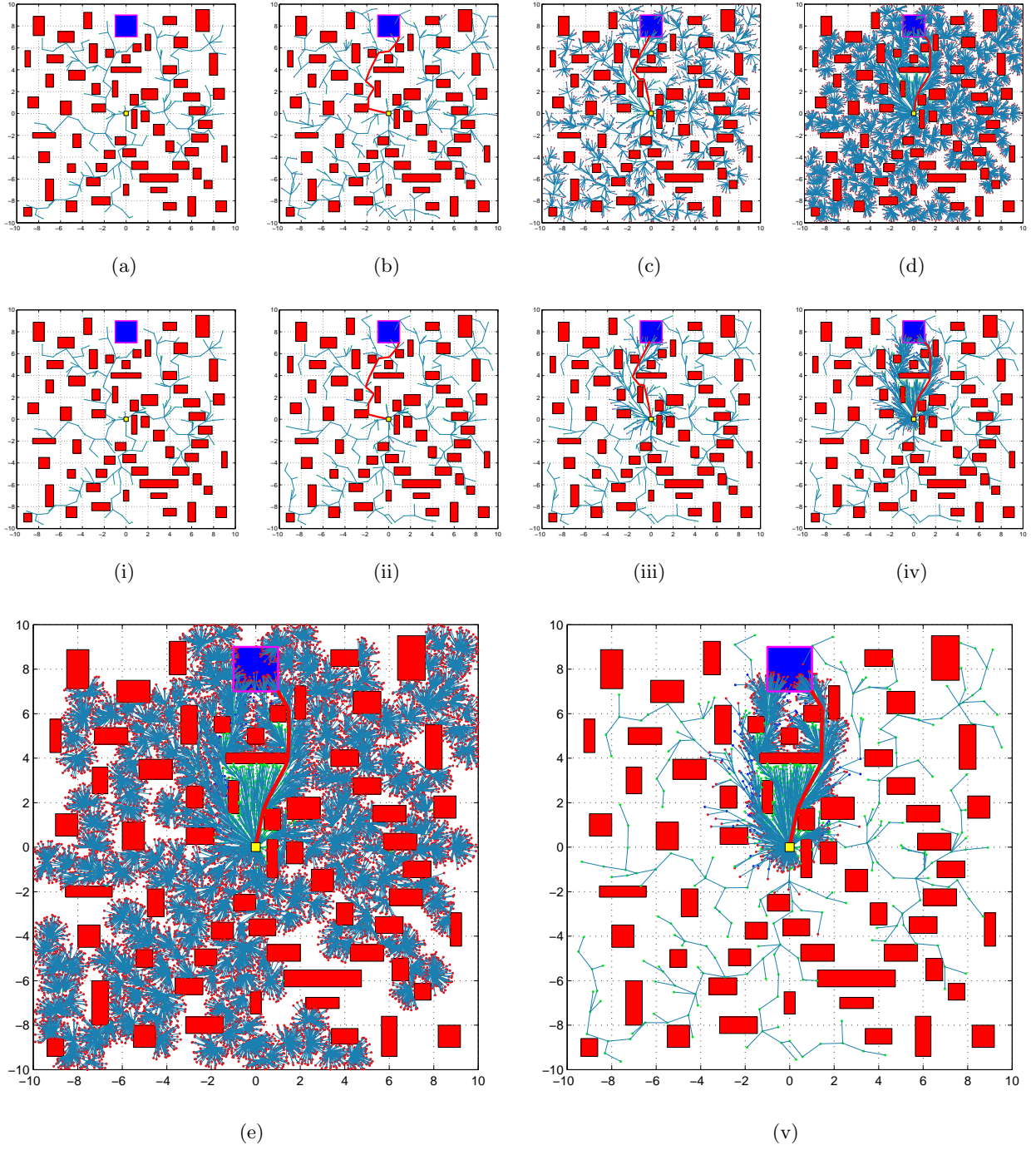


Figure 15: The evolution of the tree computed by $\text{RRT}_{V_1}^\#$ and $\text{RRT}_{V_2}^\#$ algorithms is shown in (a)-(e) and (i)-(v), respectively. The configuration of the trees (a), (i) is at 250 iterations, (b), (ii) is at 500 iterations, (c), (iii) is at 2500 iterations, (d), (iv) is at 10000 iterations, and (e), (v) is at 25000 iterations.

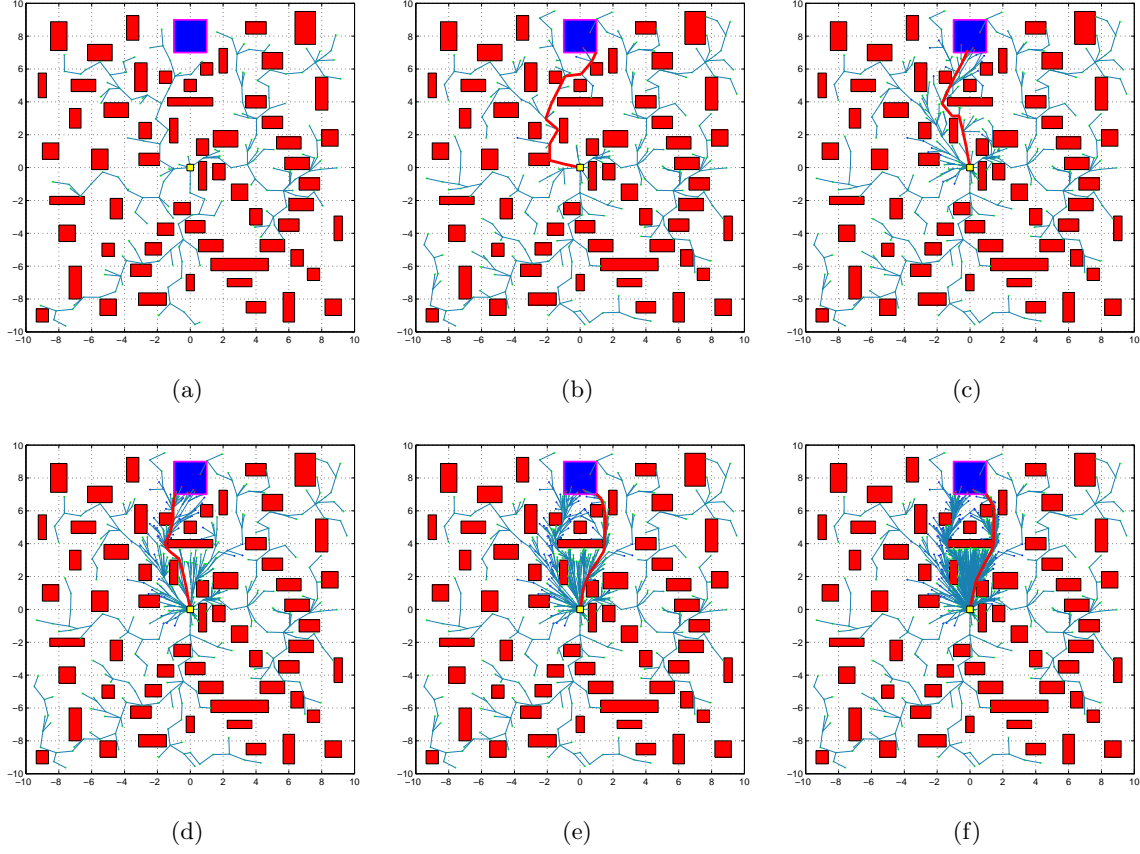


Figure 16: The evolution of the tree computed by $\text{RRT}^\#_{V_3}$ algorithm is shown in (a)-(f). The configuration of the trees in (a) is at 250 iterations, in (b) is at 500 iterations, in (c) is at 2500 iterations, in (d) is at 5000 iterations, in (e) is at 10000 iterations, and in (f) is at 25000 iterations.

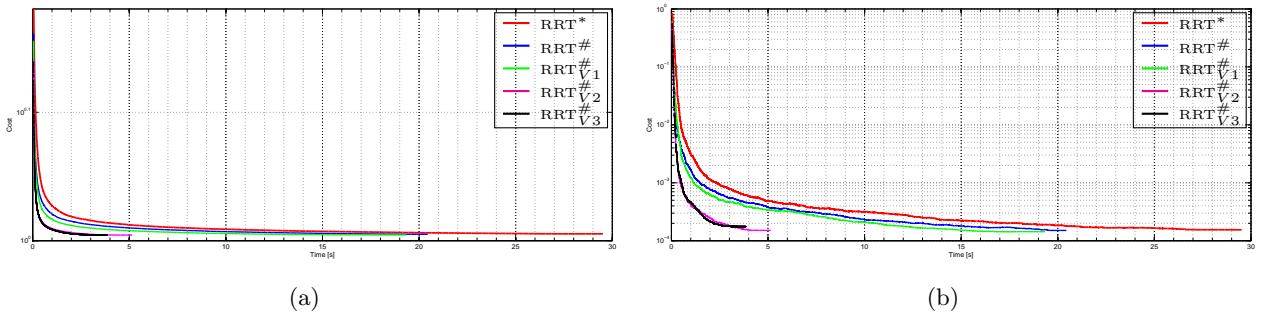


Figure 17: The change in the cost of the best paths computed by RRT^* , $\text{RRT}^\#$, and its variant algorithms and the variance in the trials are shown in (a) and (b), respectively.

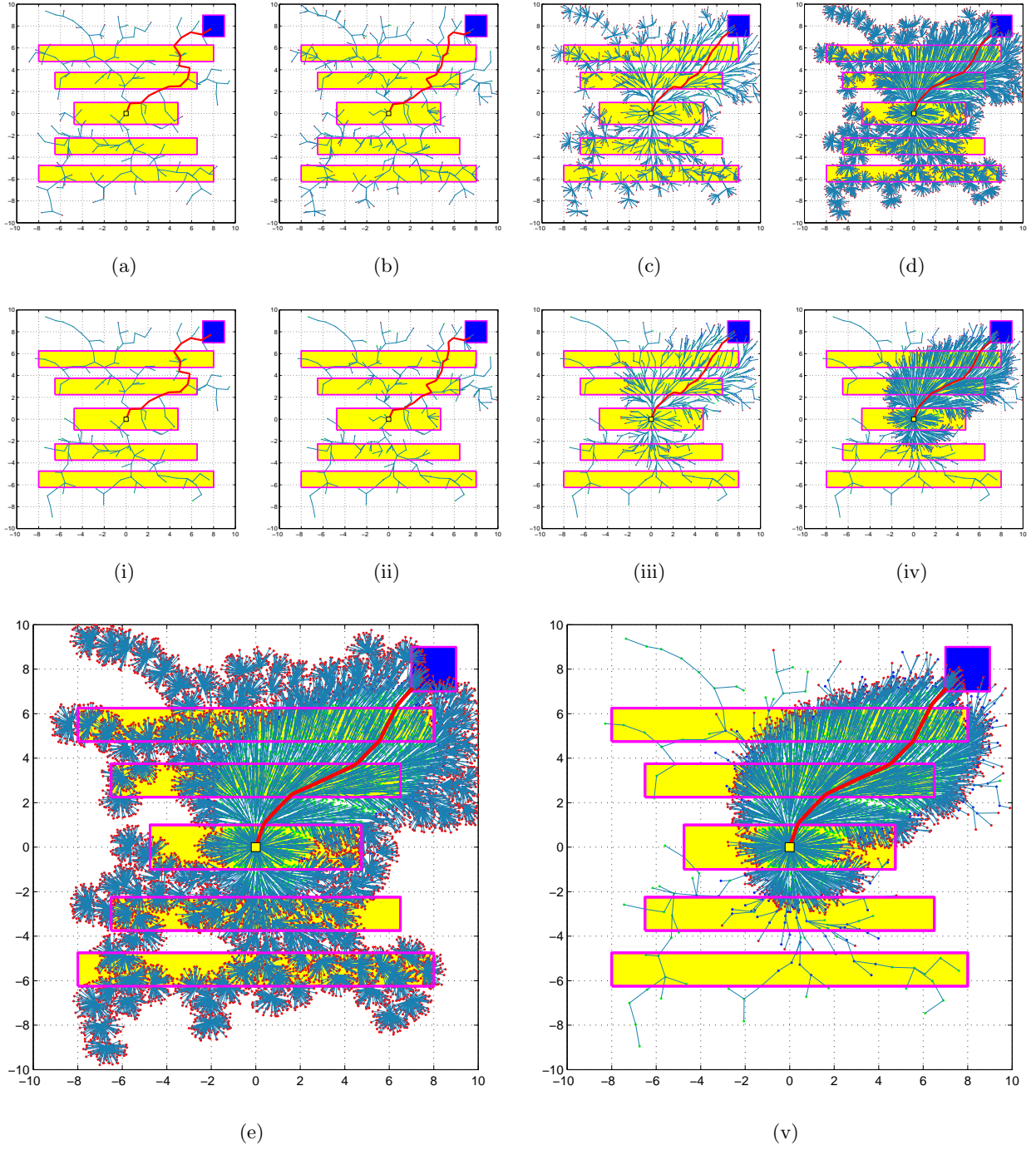


Figure 18: The evolution of the tree computed by $\text{RRT}_{V_1}^\#$ and $\text{RRT}_{V_2}^\#$ algorithms is shown in (a)-(e) and (i)-(v), respectively. The configuration of the trees (a), (i) is at 250 iterations, (b), (ii) is at 500 iterations, (c), (iii) is at 2500 iterations, (d), (iv) is at 10000 iterations, and (e), (v) is at 25000 iterations.

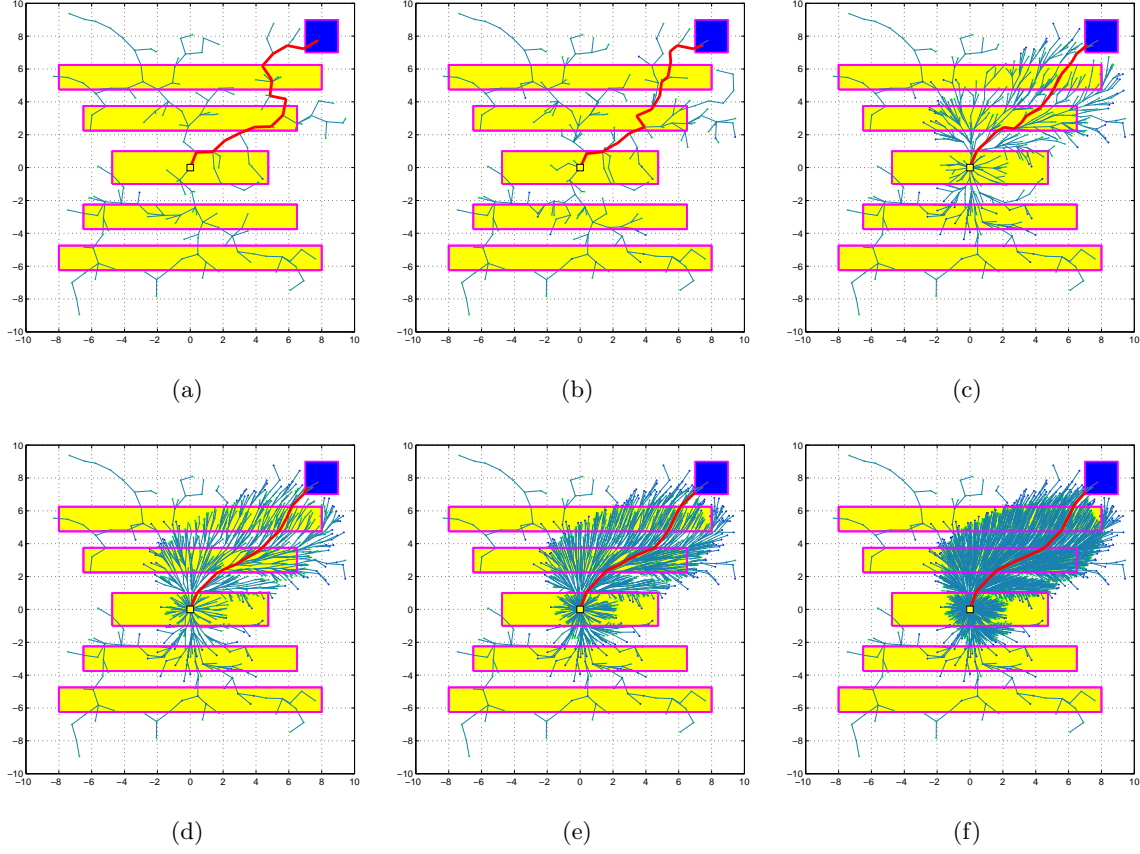


Figure 19: The evolution of the tree computed by $\text{RRT}^\#_{V_3}$ algorithm is shown in (a)-(f). The configuration of the trees in (a) is at 250 iterations, in (b) is at 500 iterations, in (c) is at 2500 iterations, in (d) is at 5000 iterations, in (e) is at 10000 iterations, and in (f) is at 25000 iterations.

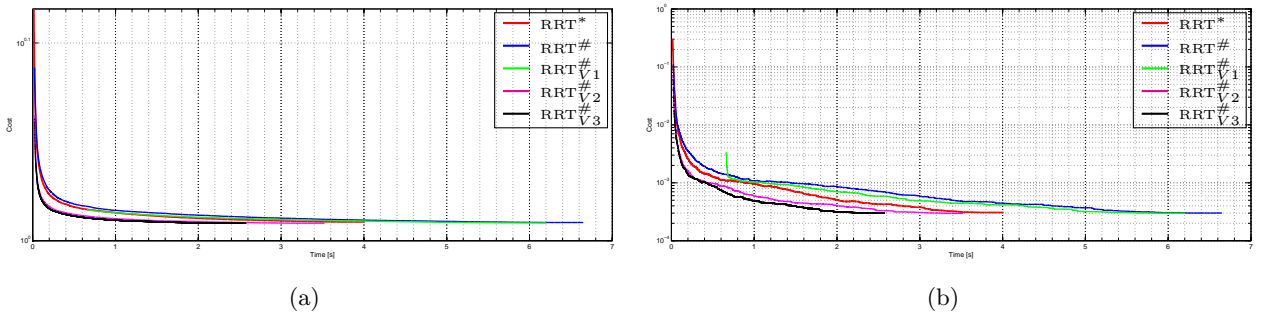


Figure 20: The change in the cost of the best paths computed by RRT^* , $\text{RRT}^\#$, and its variant algorithms and the variance in the trials are shown in (a) and (b), respectively.

A Monte-Carlo study was performed in order to compare the convergence rate and variance in the trials of all algorithms in a high dimensional search space. All algorithms were run up until 4 million iterations 100 times in a 5-dimensional search space for Problem types 1 and 2. In the second problem type, several 5-dimensional hypercubes of different size were randomly placed in the environment in order to represent obstacles. As shown in Figures 21 and 22, the $\text{RRT}_{V_2}^\#$ and $\text{RRT}_{V_3}^\#$ algorithms find the solution in a similar amount of time, and they are faster than the other algorithms. In addition, they compute solutions of lower cost than the other algorithms with smaller variance in the trials.

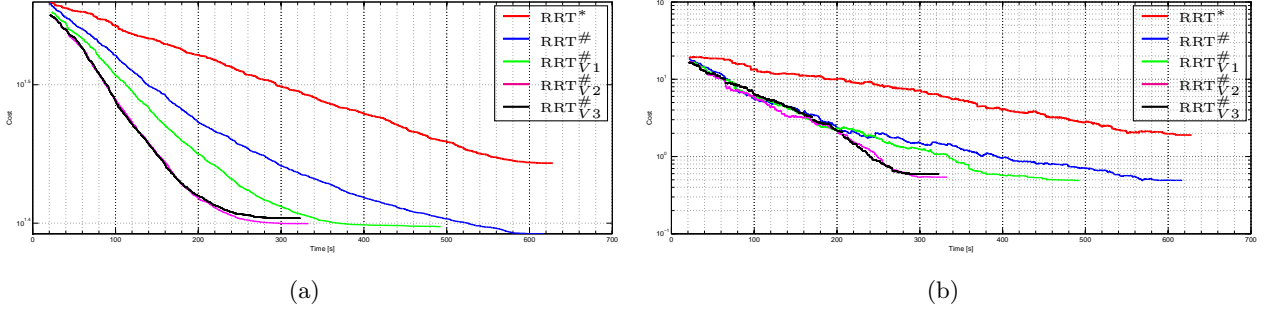


Figure 21: The change in the cost of the best paths computed by RRT^* , $\text{RRT}^\#$, and its variant algorithms and the variance of the trials is shown in (a) and (b), respectively (problem type 1, 5D search space).

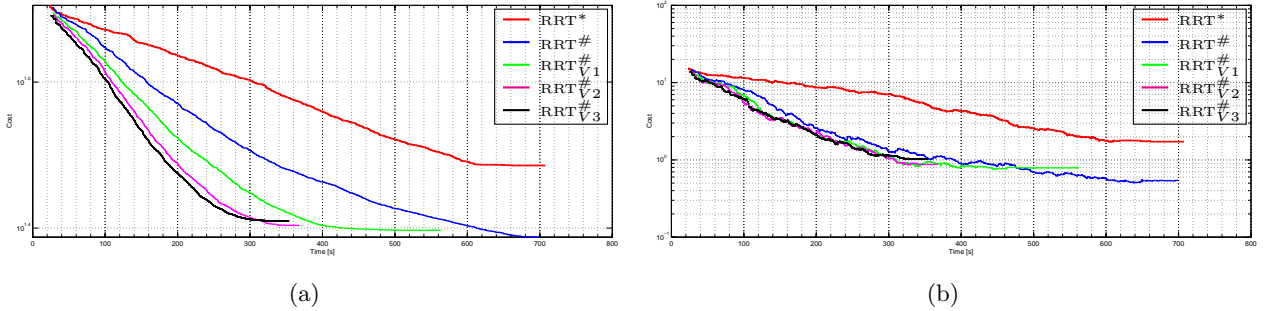


Figure 22: The change in the cost of the best paths computed by RRT^* , $\text{RRT}^\#$, and its variant algorithms and the variance of the trials are shown in (a) and (b), respectively (problem type 2, 5D search space).

The execution times of all algorithms were also compared. Results of the $\text{RRT}^\#$, $\text{RRT}_{V_1}^\#$, $\text{RRT}_{V_2}^\#$, and $\text{RRT}_{V_3}^\#$ are plotted in blue, green, magenta, and black, respectively. All algorithms were run in a 2D and a 5D environment with no obstacles for up to 750,000 and 4,000,000 iterations, respectively. The execution time of the $\text{RRT}^\#$ and its variant algorithms is normalized over that of the RRT^* algorithm and is plotted versus the number of iterations averaged over 50 trials for the 2D search space in Figure 23(a). A similar plot is also created for 100 trials in the 5D search space

and shown in Figure 23(b).

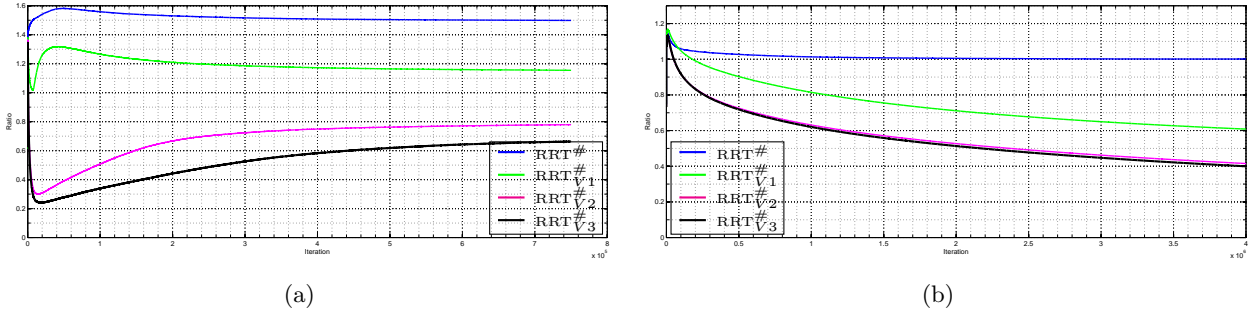


Figure 23: Comparison of execution time of all algorithms (Problem type 1)

8 Conclusion

In this paper, a new incremental sampling-based algorithm, denoted by $\text{RRT}^\#$ is presented, which offers asymptotically optimal solutions for solving motion planning problems. The $\text{RRT}^\#$ algorithm relies heavily on the random geometric graph data structure and the RRG algorithm [20], which is also known to have asymptotic optimality properties. A bottleneck of optimal sampling-based algorithms is the slow convergence to the optimal solution, although sampling-based algorithms are capable of finding a feasible solution, often almost in real-time. By incorporating consistency information of all current vertices in the tree (essentially by comparing the current cost-to-come values of the vertices with the cost-to-come values via one of the neighboring vertices) we can have more informed estimates of the optimal values of the potential paths, thus speeding up convergence. Furthermore, once a feasible path has been found, vertex consistency can be used to estimate the region where the optimal solution should be found. This results in an initial convergence rate that is better than the one of the RRT^* algorithm.

We have also introduced three variants to improve the convergence rate of the baseline $\text{RRT}^\#$ algorithm by implementing two key features: preventing the expansion of the tree towards unfavorable regions in search space, and propagating new information throughout the tree in an efficient way. The first feature allows us to limit the number of vertices in the tree, thus resulting to the algorithm running faster. The second feature allows us to compute solutions with a less number of vertices in the tree since any new information is exploited to the highest degree. As a result, the convergence rate of the baseline $\text{RRT}^\#$ can be improved significantly. Extensive numerical results have verified these observations in several simulation scenarios.

The work in this paper can be extended in several directions. First, a thorough theoretical analysis is warranted in order to provide strict bounds on the convergence rate of $\text{RRT}^\#$. Second, since $\text{RRT}^\#$ decomposes the vertex set into “promising” and “non-promising” ones, smarter sampling strategies can be developed to exploit this information. It is also crucial for the algorithm to reach the target set as early as possible in order to converge to the optimal solution faster. In that respect, a bi-directional version of the $\text{RRT}^\#$ (like the RRT-connect in [14]) can be developed in order to shorten the first time-to-connect to the goal set. Also, a parallel version of the algorithm could be implemented by running the **Extend** and **ReduceInconsistency** procedures as separate threads. A possible implementation would be to have multiple threads implementing the **Extend** procedure

and single thread implementing the `ReduceInconsistency`. Finally, the algorithm can be modified to solve motion planning problems for vehicles with complex dynamics (ground vehicles, aircraft, helicopters etc) by implementing specific local steering functions.

References

- [1] H. Choset, K. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. Kavraki, and S. Thrun. *Principles of Robot Motion: Theory, Algorithms, and Implementations*. The MIT Press, 2005.
- [2] B. Donald, P. Xavier, J. Canny, and J. Reif. Kinodynamic motion planning. *Journal of the Association for Computing Machinery*, 40(5):1048–1066, November 1993.
- [3] E. Frazzoli, M. A. Dahleh, and E. Feron. Real-time motion planning for agile autonomous vehicles. *Journal of Guidance, Control, and Dynamics*, 25(1):116–129, 2002.
- [4] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.
- [5] T. Howard and A. Kelly. Optimal rough terrain trajectory generation for wheeled mobile robots. *The International Journal of Robotics Research*, 26(2):141 – 166, February 2007.
- [6] D. Hsu, R. Kindel, J.-C. Latombe, and S. Rock. Randomized kinodynamic motion planning with moving obstacles. *International Journal of Robotics Research*, 21(3):233–255, March 2002.
- [7] S. Karaman and E. Frazzoli. Sampling-based motion planning with deterministic μ -calculus specifications. In *Decision and Control, 2009 held jointly with the 2009 28th Chinese Control Conference. CDC/CCC 2009. Proceedings of the 48th IEEE Conference on*, pages 2222–2229. IEEE, 2009.
- [8] S. Karaman and E. Frazzoli. Incremental sampling-based algorithms for optimal motion planning. In *Robotics: Science and Systems (RSS)*. Citeseer, 2010.
- [9] S. Karaman and E. Frazzoli. Sampling-based algorithms for optimal motion planning. *The International Journal of Robotics Research*, 30(7):846–894, 2011.
- [10] L. E. Kavraki and J.-C. Latombe. Randomized preprocessing of configuration space for fast path planning. Technical Report STAN-CS-93-1490, Dept. Computer Science, Stanford University, Stanford, CA, 1993.
- [11] L. E. Kavraki, P. Švestka, J.-C. Latombe, and M. H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12(4):566–580, 1996.
- [12] L.E. Kavraki, P. Svestka, J.C. Latombe, and M.H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *Robotics and Automation, IEEE Transactions on*, 12(4):566–580, 1996.
- [13] S. Koenig, M. Likhachev, and D. Furcy. Lifelong planning A*. *Artificial Intelligence*, 155(1-2):93–146, 2004.
- [14] J.J. Kuffner Jr. and S.M. LaValle. RRT-connect: An efficient approach to single-query path planning. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, volume 2, pages 995–1001. IEEE, 2000.

- [15] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006.
- [16] S. M. LaValle and J. J. Kuffner. Rapidly-exploring random trees: Progress and prospects. In B. R. Donald, K. Lynch, and D. Rus, editors, *New Directions in Algorithmic and Computational Robotics*, pages 293–308. AK Peters, 2001.
- [17] S. M. LaValle and J.J. Kuffner. Randomized kinodynamic planning. *The International Journal of Robotics Research*, 20(5):378, 2001.
- [18] N.J. Nilsson. Problem-solving methods in artificial intelligence. 1971.
- [19] J. Pearl. Heuristics: intelligent search strategies for computer problem solving. 1984.
- [20] M. Penrose. *Random geometric graphs*, volume 5. Oxford University Press, USA, 2003.
- [21] E. Plaku, L. E. Kavraki, and M. Y. Vardi. Motion planning with dynamics by a synergistic combination of layers of planning. *IEEE Transactions on Robotics*, 26(3):469–482, 2010.
- [22] P. Švestka. A probabilistic approach to motion planning for car-like robots. Technical Report RUU-CS-1993-18, Dept. Computer Science, Utrecht University, Utrecht, The Netherlands, 1993.